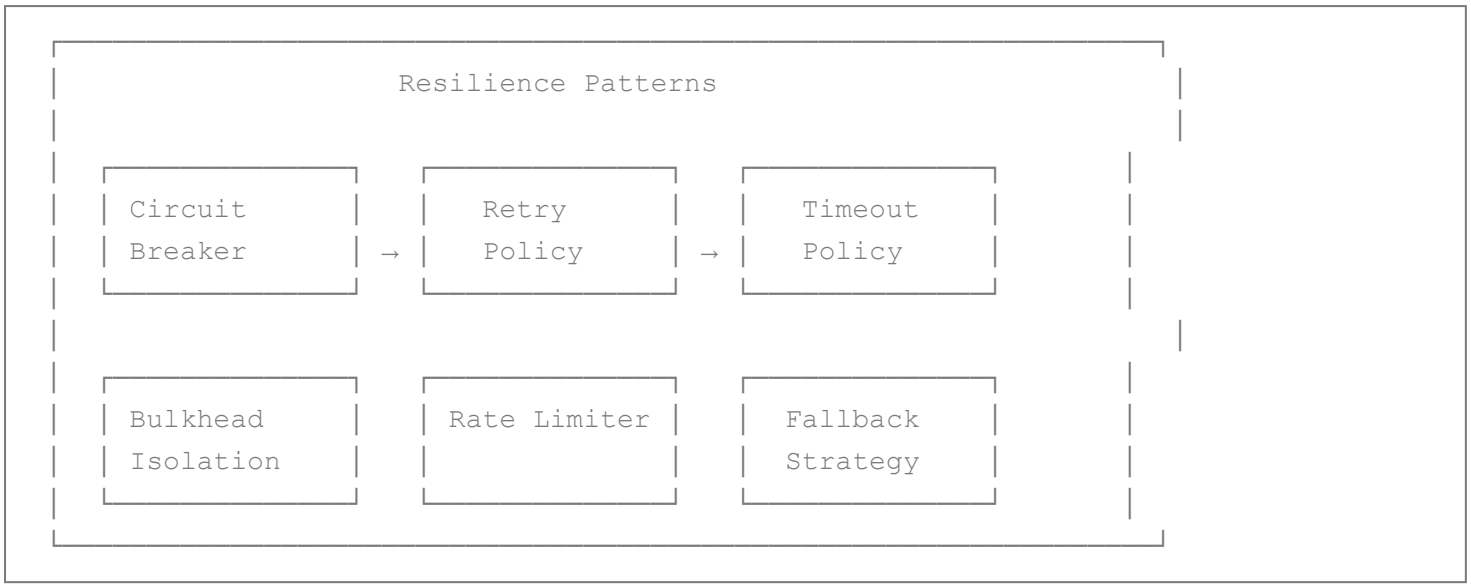
Resilience Strategy - Patterns & Configuration

Version: 1.0
Date: 2025-11-26
Status: Implementation Ready
Library: Resilience4j 2.1.0

---

1. Resilience Overview



---

2. Timeout Strategy

2.1 Timeout Configuration

```yaml
# application.yml
resilience4j:
  timelimiter:
    instances:
      # External HTTP calls (providers)
      externalHttp:
        timeoutDuration: 5s
        cancelRunningFuture: true

      # Internal HTTP calls (microservices)
      internalHttp:
        timeoutDuration: 3s
        cancelRunningFuture: true

      # JDBC operations
      database:
```

```
        timeoutDuration: 2s
        cancelRunningFuture: false

      # Kafka operations
      kafka:
        timeoutDuration: 1500ms
        cancelRunningFuture: true
```

2.2 Implementation

```
@Service
public class TwilioSmsProvider implements SignatureProvider {

    @TimeLimiter(name = "externalHttp", fallbackMethod = "sendFallback")
    public CompletableFuture<ProviderResult> sendAsync(SignatureChallenge
challenge) {
        return CompletableFuture.supplyAsync(() -> {
            // Twilio API call with 5s timeout
            return twilioClient.messages().create(
                new Message.Builder()
                    .to(challenge.getRecipient())
                    .body(challenge.getMessage())
                    .build()
            );
        });
    }

    private CompletableFuture<ProviderResult> sendFallback(
        SignatureChallenge challenge,
        TimeoutException ex
    ) {
        log.error("Twilio timeout after 5s: {}", ex.getMessage());
        return CompletableFuture.completedFuture(
            ProviderResult.failure("TIMEOUT", "Provider timeout exceeded")
        );
    }
}
```

---

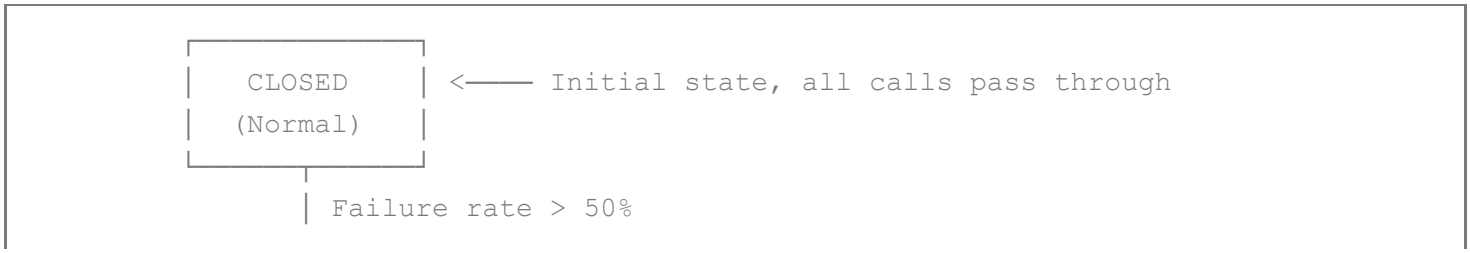
3. Circuit Breaker Pattern

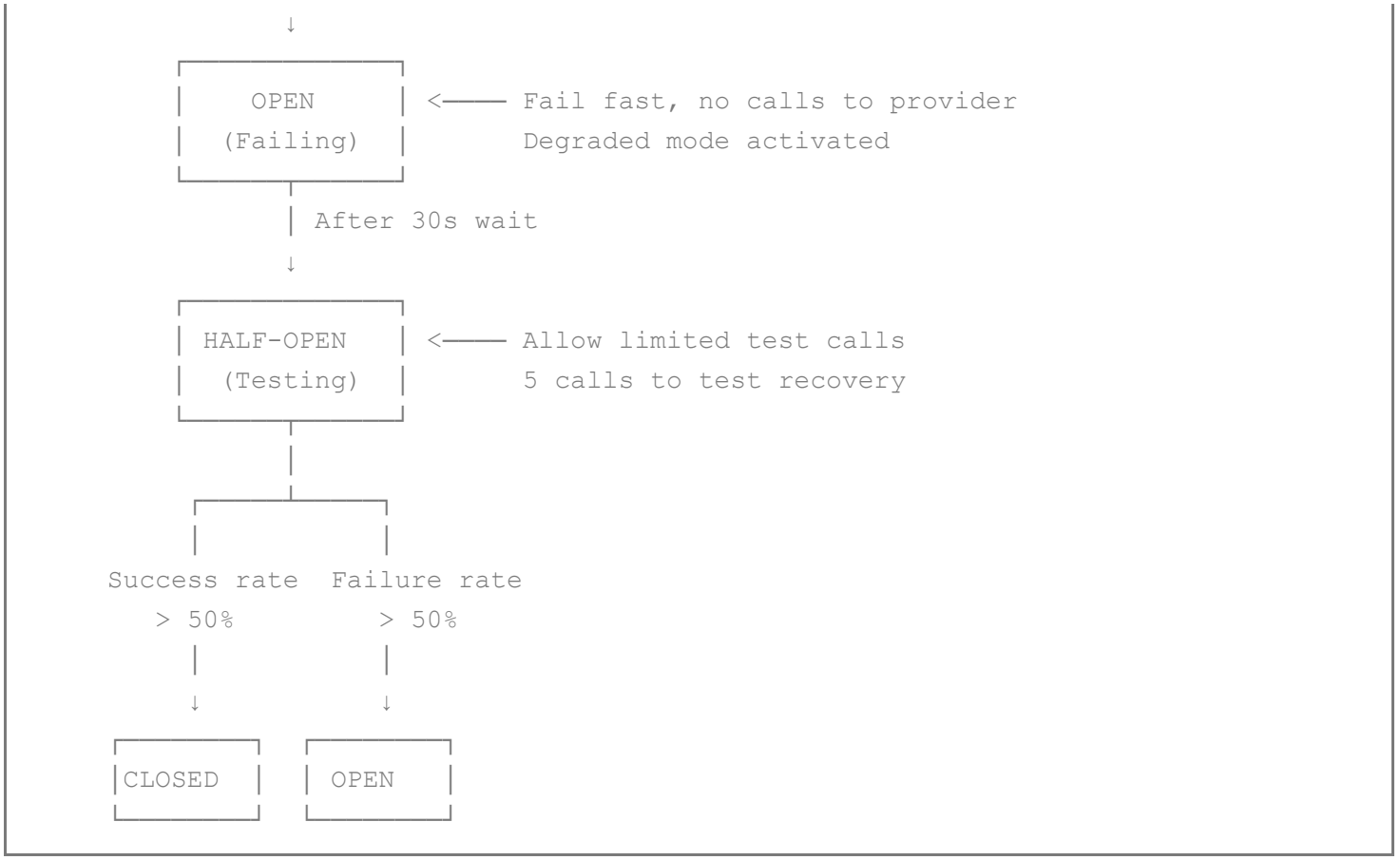
3.1 Circuit Breaker Configuration

```
resilience4j:
  circuitbreaker:
    instances:
      # Provider circuit breakers
```

```yaml
    twilioProvider:
      registerHealthIndicator: true
      slidingWindowType: COUNT_BASED
      slidingWindowSize: 100
      minimumNumberOfCalls: 10
      failureRateThreshold: 50          # Open if >50% fail
      slowCallRateThreshold: 50          # Open if >50% slow
      slowCallDurationThreshold: 3s      # Call considered slow if >3s
      waitDurationInOpenState: 30s       # Wait before half-open
      permittedNumberOfCallsInHalfOpenState: 5
      automaticTransitionFromOpenToHalfOpenEnabled: true
      recordExceptions:
        - java.io.IOException
        - java.util.concurrent.TimeoutException
        - com.twilio.exception.ApiException
      ignoreExceptions:
        - com.bank.signature.domain.exception.ValidationException

    pushProvider:
      registerHealthIndicator: true
      slidingWindowSize: 100
      minimumNumberOfCalls: 10
      failureRateThreshold: 50
      waitDurationInOpenState: 30s

    voiceProvider:
      registerHealthIndicator: true
      slidingWindowSize: 100
      minimumNumberOfCalls: 10
      failureRateThreshold: 50
      waitDurationInOpenState: 30s

    # Database circuit breaker
    database:
      registerHealthIndicator: true
      slidingWindowSize: 20
      minimumNumberOfCalls: 5
      failureRateThreshold: 50
      waitDurationInOpenState: 10s
```

3.2 Circuit Breaker States





3.3 Implementation

```
@Component
public class SignatureProviderAdapter implements SignatureProviderPort {

    private final Map<ProviderType, SignatureProvider> providers;
    private final CircuitBreakerRegistry circuitBreakerRegistry;
    private final DegradedModeManager degradedModeManager;

    @Override
    public ProviderResult sendChallenge(SignatureChallenge challenge) {
        String providerName = challenge.getProvider().name().toLowerCase() +
"Provider";
        CircuitBreaker circuitBreaker =
circuitBreakerRegistry.circuitBreaker(providerName);

        // Register event listeners
        circuitBreaker.getEventPublisher()
            .onStateTransition(this::onCircuitBreakerStateChange)
            .onError(this::onCircuitBreakerError);

        // Execute with circuit breaker protection
        return circuitBreaker.executeSupplier(() -> {
            SignatureProvider provider = providers.get(challenge.getProvider());
            return provider.send(challenge);
```

```
        });
    }

    private void onCircuitBreakerStateChange(CircuitBreakerOnStateTransitionEvent
event) {
        String providerName = event.getCircuitBreakerName();
        CircuitBreaker.State newState = event.getStateTransition().getToState();

        log.warn("Circuit breaker {} transitioned to {}", providerName, newState);

        if (newState == CircuitBreaker.State.OPEN) {
            // Activate degraded mode
            degradedModeManager.activateDegradedMode(
                providerName,
                Duration.ofMinutes(5)
            );

            // Publish event
            eventPublisher.publish(new ProviderFailed(
                providerName,
                event.getMetrics().getFailureRate(),
                true  // degraded mode
            ));
        } else if (newState == CircuitBreaker.State.CLOSED) {
            // Deactivate degraded mode
            degradedModeManager.deactivateDegradedMode(providerName);
        }
    }
}
```

---

4. Retry Strategy

4.1 Retry Configuration

```yaml
resilience4j:
  retry:
    instances:
      # Provider retries (transient failures)
      providerRetry:
        maxAttempts: 3
        waitDuration: 500ms
        exponentialBackoffMultiplier: 2    # 500ms, 1s, 2s
        retryExceptions:
          - java.net.ConnectException
          - java.net.SocketTimeoutException
          - org.springframework.web.client.ResourceAccessException
```

```yaml
      ignoreExceptions:
        - com.bank.signature.domain.exception.ValidationException
        - javax.validation.ValidationException

    # Database retries (deadlock, connection issues)
    databaseRetry:
      maxAttempts: 2
      waitDuration: 100ms
      retryExceptions:
        - org.springframework.dao.TransientDataAccessException
        - java.sql.SQLException

    # Kafka retries
    kafkaRetry:
      maxAttempts: 3
      waitDuration: 200ms
      exponentialBackoffMultiplier: 2
```

4.2 Retry with Fallback

```java
@Service
public class RetryWithFallbackUseCaseImpl implements RetryWithFallbackUseCase {

    @Retry(name = "providerRetry", fallbackMethod = "fallbackToNextChannel")
    @Override
    public ProviderResult attemptChallenge(SignatureRequest request, ChannelType
channel) {
        // Attempt to send challenge
        SignatureChallenge challenge = request.createChallenge(channel);
        ProviderResult result = providerPort.sendChallenge(challenge);

        if (!result.isSuccess()) {
            throw new ProviderException("Provider failed: " +
result.getErrorCode());
        }

        return result;
    }

    private ProviderResult fallbackToNextChannel(
        SignatureRequest request,
        ChannelType currentChannel,
        Exception ex
    ) {
        log.warn("All retries exhausted for channel {}, attempting fallback",
currentChannel, ex);
```

```
        // Determine next fallback channel
        ChannelType fallbackChannel =
fallbackStrategy.getNextChannel(currentChannel);

        if (fallbackChannel == null) {
            // No more fallbacks
            throw new FallbackExhaustedException("All channels exhausted");
        }

        // Recursive retry with new channel
        return attemptChallenge(request, fallbackChannel);
    }
}
```

---

5. Fallback Chain Strategy

5.1 Fallback Sequence

```
Attempt 1: PUSH (cheapest, best UX)
   ↓ FAILED
Attempt 2: SMS (reliable, moderate cost)
   ↓ FAILED
Attempt 3: VOICE (most expensive, highest success rate)
   ↓ FAILED
Result: SIGNATURE_FAILED
```

5.2 Fallback Implementation

```
@Service
public class FallbackStrategyService {

    private static final Map<ChannelType, ChannelType> FALLBACK_CHAIN = Map.of(
        ChannelType.PUSH, ChannelType.SMS,
        ChannelType.SMS, ChannelType.VOICE,
        ChannelType.VOICE, null  // No more fallbacks
    );

    public ChannelType getNextChannel(ChannelType current) {
        return FALLBACK_CHAIN.get(current);
    }

    public boolean hasMoreFallbacks(ChannelType current) {
        return FALLBACK_CHAIN.get(current) != null;
    }

    public List<ChannelType> getRemainingFallbacks(ChannelType current) {
```

```
        List<ChannelType> remaining = new ArrayList<>();
        ChannelType next = current;

        while ((next = getNextChannel(next)) != null) {
            remaining.add(next);
        }

        return remaining;
    }
}
```

5.3 Fallback Rules

```
@Service
public class FallbackDecisionService {

    public boolean shouldAttemptFallback(SignatureRequest request, ProviderResult
result) {
        // Rule 1: No re-evaluation of routing rules in fallback
        // Rule 2: One active challenge at a time
        // Rule 3: Respect provider degraded mode
        // Rule 4: Check if error is retryable

        if (!result.isRetryable()) {
            log.info("Error is not retryable, skipping fallback");
            return false;
        }

        if (request.getChallenges().size() >= MAX_ATTEMPTS) {
            log.warn("Max attempts ({}) reached, no more fallbacks",
MAX_ATTEMPTS);
            return false;
        }

        ChannelType currentChannel =
request.getActiveChallenge().getChannelType();
        if (!fallbackStrategy.hasMoreFallbacks(currentChannel)) {
            log.info("No more fallback channels available");
            return false;
        }

        return true;
    }
}
```

---

6. Bulkhead Isolation

6.1 Thread Pool Isolation

```yaml
resilience4j:
  bulkhead:
    instances:
      # Isolate provider calls to prevent thread pool exhaustion
      providerBulkhead:
        maxConcurrentCalls: 50
        maxWaitDuration: 100ms

  thread-pool-bulkhead:
    instances:
      twilioProvider:
        maxThreadPoolSize: 10
        coreThreadPoolSize: 5
        queueCapacity: 50
        keepAliveDuration: 20ms

      pushProvider:
        maxThreadPoolSize: 10
        coreThreadPoolSize: 5
        queueCapacity: 50

      voiceProvider:
        maxThreadPoolSize: 5
        coreThreadPoolSize: 2
        queueCapacity: 20
```

6.2 Implementation

```java
@Component
public class TwilioSmsProvider implements SignatureProvider {

    @Bulkhead(name = "providerBulkhead", type = Bulkhead.Type.THREADPOOL)
    @Override
    public CompletableFuture<ProviderResult> sendAsync(SignatureChallenge
challenge) {
        return CompletableFuture.supplyAsync(() -> {
            // Isolated in separate thread pool
            return twilioClient.send(challenge);
        });
    }
}
```

Benefit: If Twilio is slow/hanging, only its thread pool saturates, not the entire application.

7. Rate Limiting

7.1 Rate Limiter Configuration

```yaml
resilience4j:
  ratelimiter:
    instances:
      # Protect against abuse
      signatureCreation:
        limitForPeriod: 100
        limitRefreshPeriod: 1s
        timeoutDuration: 0s  # Fail immediately if limit exceeded

      # Admin operations
      ruleManagement:
        limitForPeriod: 10
        limitRefreshPeriod: 1s
        timeoutDuration: 0s
```

7.2 Implementation

```java
@RestController
@RequestMapping("/api/v1/signatures")
public class SignatureController {

    @PostMapping
    @RateLimiter(name = "signatureCreation")
    public ResponseEntity<SignatureResponse> createSignature(
        @RequestHeader("Idempotency-Key") String idempotencyKey,
        @RequestBody CreateSignatureRequest request
    ) {
        // Rate limited to 100 req/s
        SignatureResponse response = startUseCase.start(request);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }
}
```

---

8. Degraded Mode Management

8.1 Degraded Mode State Machine

```java
@Service
public class DegradedModeManager {

    private final ConnectorConfigRepository connectorRepo;
```

```
    private final EventPublisherPort eventPublisher;

    public void activateDegradedMode(String provider, Duration pauseDuration) {
        ConnectorConfig config = connectorRepo.findByProvider(provider)
            .orElseThrow(() -> new IllegalArgumentException("Provider not found: "
+ provider));

        // Mark provider as degraded
        config.setDegradedMode(true);
        config.setDegradedSince(Instant.now());
        connectorRepo.save(config);

        log.error("Provider {} entered DEGRADED MODE for {}", provider,
pauseDuration);

        // Publish event
        eventPublisher.publish(new ProviderFailed(
            provider,
            config.getErrorRate(),
            true,
            Instant.now().plus(pauseDuration)
        ));

        // Schedule re-enablement
        scheduledExecutor.schedule(
            () -> deactivateDegradedMode(provider),
            pauseDuration.toMillis(),
            TimeUnit.MILLISECONDS
        );
    }

    public void deactivateDegradedMode(String provider) {
        ConnectorConfig config = connectorRepo.findByProvider(provider)
            .orElseThrow(() -> new IllegalArgumentException("Provider not found: "
+ provider));

        config.setDegradedMode(false);
        config.setDegradedSince(null);
        config.setErrorRate(BigDecimal.ZERO);
        connectorRepo.save(config);

        log.info("Provider {} recovered from DEGRADED MODE", provider);
    }

    public boolean isProviderAvailable(String provider) {
        return connectorRepo.findByProvider(provider)
            .map(config -> config.isEnabled() && !config.isDegradedMode())
```

```
                .orElse(false);
        }
    }
```

8.2 Provider Health Monitoring

```
@Component
@Slf4j
public class ProviderHealthMonitor {

    private final ConnectorConfigRepository connectorRepo;
    private final DegradedModeManager degradedModeManager;

    @Scheduled(fixedRate = 60000)  // Every 1 minute
    public void checkProviderHealth() {
        List<ConnectorConfig> connectors = connectorRepo.findAllEnabled();

        for (ConnectorConfig connector : connectors) {
            double errorRate = calculateErrorRate(connector);
            connector.setErrorRate(BigDecimal.valueOf(errorRate));
            connector.setLastHealthCheck(Instant.now());

            // Circuit breaker threshold: 50%
            if (errorRate > 50.0 && !connector.isDegradedMode()) {
                log.warn("Provider {} error rate {}% exceeds threshold",
                    connector.getProvider(), errorRate);

                degradedModeManager.activateDegradedMode(
                    connector.getProvider(),
                    Duration.ofMinutes(5)
                );
            }

            connectorRepo.save(connector);
        }
    }

    private double calculateErrorRate(ConnectorConfig connector) {
        // Query last 100 challenges for this provider
        List<SignatureChallenge> recentChallenges =

 challengeRepo.findTop100ByProviderOrderByCreatedAtDesc(connector.getProvider());

        if (recentChallenges.isEmpty()) {
            return 0.0;
        }
```

```
        long failedCount = recentChallenges.stream()
            .filter(c -> c.getStatus() == ChallengeStatus.FAILED)
            .count();

        return (failedCount * 100.0) / recentChallenges.size();
    }
}
```

---

9. Idempotency Guarantees

9.1 Idempotency Key Filter

```
@Component
@Order(1)
public class IdempotencyFilter extends OncePerRequestFilter {

    private final IdempotencyRepository idempotencyRepo;

    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain
    ) throws ServletException, IOException {

        if (!"POST".equals(request.getMethod())) {
            filterChain.doFilter(request, response);
            return;
        }

        String idempotencyKey = request.getHeader("Idempotency-Key");
        if (idempotencyKey == null) {
            response.sendError(400, "Missing Idempotency-Key header");
            return;
        }

        // Check if request already processed
        Optional<IdempotencyRecord> existing =
            idempotencyRepo.findByKey(idempotencyKey);

        if (existing.isPresent()) {
            // Return cached response
            IdempotencyRecord record = existing.get();

            if (record.isExpired()) {
                // Key expired (24h), allow reuse
```

```
                idempotencyRepo.delete(record);
            } else {
                // Return cached response
                response.setStatus(record.getStatusCode());
                response.setContentType("application/json");
                response.getWriter().write(record.getResponseBody());
                return;
            }
        }

        // Wrap response to capture output
        ContentCachingResponseWrapper responseWrapper =
            new ContentCachingResponseWrapper(response);

        filterChain.doFilter(request, responseWrapper);

        // Cache response for 24h
        String responseBody = new String(
            responseWrapper.getContentAsByteArray(),
            responseWrapper.getCharacterEncoding()
        );

        idempotencyRepo.save(new IdempotencyRecord(
            idempotencyKey,
            responseWrapper.getStatus(),
            responseBody,
            Instant.now().plus(Duration.ofHours(24))
        ));

        responseWrapper.copyBodyToResponse();
    }
}
```

---

10. Graceful Shutdown

```
server:
  shutdown: graceful

spring:
  lifecycle:
    timeout-per-shutdown-phase: 30s
```

```
@Component
public class GracefulShutdownListener {

    private final SignatureRequestRepository signatureRepo;
```

```
    @PreDestroy
    public void onShutdown() {
        log.info("Graceful shutdown initiated...");

        // 1. Stop accepting new requests (Spring Boot handles this)

        // 2. Wait for in-flight requests to complete (max 30s)

        // 3. Mark pending signatures as ABORTED
        List<SignatureRequest> pending =
signatureRepo.findByStatus(SignatureStatus.PENDING);

        for (SignatureRequest request : pending) {
            request.abort(AbortReason.SYSTEM_SHUTDOWN);
            signatureRepo.save(request);
        }

        log.info("Graceful shutdown completed");
    }
}
```

---

11. Testing Resilience

11.1 Chaos Engineering Tests

```
@SpringBootTest
class ResilienceIT {

    @Test
    void shouldHandleProviderTimeout() {
        // Given: Mock provider with 10s delay (exceeds 5s timeout)
        when(twilioClient.send(any())).thenAnswer(invocation -> {
            Thread.sleep(10000);
            return ProviderResult.success("never-reached");
        });

        // When
        assertThrows(TimeoutException.class, () -> {
            signatureService.create(createRequest());
        });

        // Then: Should trigger fallback to SMS
        verify(fallbackService).attemptFallback(any(), eq(ChannelType.PUSH));
    }
```

```
    @Test
    void shouldOpenCircuitBreakerAfter50PercentFailures() {
        // Given: Provider fails 60 out of 100 calls
        for (int i = 0; i < 60; i++) {
            when(twilioClient.send(any())).thenThrow(new IOException("Network
error"));
            assertThrows(ProviderException.class, () ->
providerPort.sendChallenge(challenge));
        }

        // Then: Circuit breaker should be OPEN
        CircuitBreaker cb =
circuitBreakerRegistry.circuitBreaker("twilioProvider");
        assertThat(cb.getState()).isEqualTo(CircuitBreaker.State.OPEN);

        // And: Provider should be in degraded mode
        ConnectorConfig config = connectorRepo.findByProvider("TWILIO").get();
        assertThat(config.isDegradedMode()).isTrue();
    }
}
```

Status: ✅ **COMPLETE - READY FOR CONFIGURATION**

Next Steps:

- Configure Resilience4j in Spring Boot
- Implement provider-specific fallback logic
- Set up alerting for circuit breaker events
- Test with chaos engineering (Chaos Monkey, Toxiproxy)